Interval Arithmetic: an efficient implementation and an application to computational geometry

Sylvain Pion

INTERVAL ARITHMETIC: AN EFFICIENT IMPLEMENTATION AND AN APPLICATION TO COMPUTATIONAL GEOMETRY *

Sylvain PION

INRIA Sophia-Antipolis, BP 93,
06902 Sophia-Antipolis cedex, France.
Sylvain.Pion@sophia.inria.fr

Abstract

We discuss interval techniques for speeding up the exact evaluation of geometric predicates and describe a C++ implementation of interval arithmetic that is strongly influenced by the rounding modes of the widely used IEEE 754 standard. Using this approach we engineer an efficient floating point filter for the computation of geometric predicates. We validate our approach experimentally, comparing it with other static, dynamic and semi-static filters.

Keywords: Interval arithmetic, computational geometry, robustness, dynamic filters, C++ implementation.

1 Introduction

Computational geometry is a branch of computer science that deals with building combinatorial structures from geometric objects (stored numerically). Examples of objects are points and lines, and examples of structures are convex hulls and triangulations. The algorithms computing these structures appear to raise particular robustness problems, because the combinatorial part is built from decisions based on the relative position of geometric objects. From the implementation point of view, the bridge between the numerical and the combinatorial parts is well identified in specific functions called predicates (Fig. 1). These predicates take a few geometric objects as input (typically points with there coordinates), and return a boolean or a multi-state type (typically an *enum* type, like {-1,0,1}).

Though an approximate evaluation (using double precision floating point numbers) of the predicates might give a bad result less than once over a million times, this is critical, because it will break the combinatorial part relying on the exactness of the predicates, which in turn can make the algorithm crash or fall into an endless loop. So the robustness of most geometric algorithms boils down to the exactness of those predicates.

We have to deal with these problems in the implementation of the CGAL C++ library, which is a project gathering geometric algorithms. Therefore we are looking for generic and efficient solutions.

There are numerous approaches to get the problem under control, of which the immediate solution of exact computation stands out because of its generality (using a multiprecision integer

*This research was partially supported by the ESPRIT IV LTR Project No. 28155 (GALIA).

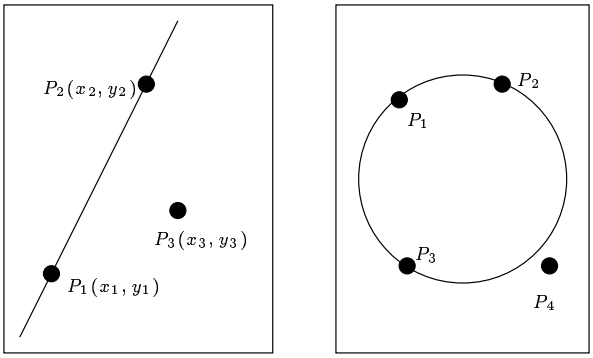

Figure 1: *Simple geometric predicates : the* **orientation** *and the* **in_sphere** *tests. These functions test whether a point lies on the left or right side of a line, and in or out a circle. They reduce to signs of simple polynomials, based on low dimensional determinants.*

package or such). For faster yet exact computation, arithmetic filters were proposed in [10, 15] and showed to be efficient both in practice [11, 18] and in theory [9]. They basically consist in a fast first step to the evaluation of the predicates that controls the roundoff error, so that the expensive exact numerical computation is only needed in rare cases.

The work done so far mainly concerns static and semi–static filters where the error bounds, or at least parts of it, are determined at compile time. Static filters are restricted to integral expressions of small bounded depth and, which is even worse, require that good upper bounds on the input variables are known in advance. Semi-static filters remedy most of the mentioned problems, but divisions and square roots can only be handled at the price of a significantly reduced quality of the resulting error bounds. Moreover, in order to use semi-static filters, the complete structure of a computation has to be preprocessed in advance. Dynamic filtering on the other hand is often considered too inefficient for use in computational geometry. First advances have been made to incorporate dynamic filters into the LEDA reals [6].

In this paper, we propose to use *interval arithmetic* [16, 17, 12, 14] for more efficient dynamic filters. The technique is based on carefully engineered interval arithmetic, which is sometimes better than the usual method found e.g. in the PROFIL/BIAS library [20], which requires frequent rounding mode changes. It is very simple to use and yields the most flexible dynamic floating-point filters we know: divisions can be handled as well as square roots and hence the technique is not limited to rational expressions. With the IEEE 754 standard for floating point computations [13], the computed intervals are locally optimal in the sense that every single operation results in the smallest possible interval. Consequently, the produced filters have the maximal achievable probability of success. On the other hand, interval arithmetic is still relatively fast, being roughly 3-8 times slower than floating-point evaluation. Our implementation of interval arithmetic is heavily influenced by that of the rounding modes of the IEEE 754 standard.

There are two intrinsic limitations to the use of interval arithmetic. First, it may fail to detect degenerate instances (e.g. aligned points for the orientation test predicate) because of the roundoff errors. When the operations are performed exactly, the interval is reduced to a point and this certifies that there is indeed a degeneracy. However, this only happens when the bit length of the input data is small with respect to the machine precision. Second, although the computed intervals are optimal for every single operation, there can be a huge overestimation of the error for a cascaded sequence of operations. However, concerning geometric predicates, the corresponding expressions are usually not very deeply cascaded, so that the probability of success is still very high, as shown in [9] for a few common examples.

Our paper is organized as follows. In section 2 we lay out a classification of filters into

static, semi-static, and dynamic filters and we discuss their usage in precompiled, hand-coded and fully packaged cascaded computation. In section 3 we introduce the basic notions of interval arithmetic and give an overview of the efficient methods to obtain verified inclusions. Here we introduce a heuristic measure of the effectiveness of interval analysis for a given expression $\mathcal{E}$. In section 4 the approach is validated experimentally with an implementation of interval arithmetic that relies on the symmetry between the rounding modes of the IEEE 754 standard.

2 Arithmetic filters

When we want to compute exactly the sign of an arithmetic expression $\mathcal{E} = \mathcal{E}(x_1, \ldots, x_n)$ (such as a geometric predicate in our case), the first generic solution that comes to mind is the use of an exact number type (such as those provided by many multiprecision packages). However, this solution rarely has a negligible cost, because geometric applications tend to make a lot of calls to such functions.

On the other hand, considering that these functions appear to be evaluated exactly most of the time when using a fast number type like double precision floating point numbers, we would like to adopt a two-time scheme that first computes the expression using a fast type, controlling the error done while doing so, then either :

success case: returns the result when it is certified to be exact (the error done is small enough so that the sign can be determined exactly), or

failure case: asks to recompute more precisely (with an exact number type) if the first fast computation failed to give an exact certified result.

This is what we call an arithmetic filter.

We will focus on single precision[1] floating point filters, because they have a speed comparable to the simple floating point evaluation. We distinguish mainly three kinds of such filters, described below.

Fully static: An upper bound $|x_i| \leq X_i$ is known for each i, and $\mathcal{E}$ uses only $+, -, \cdot, \sqrt{}$. Then E is computed such that the error on computing $\mathcal{E}$ is bounded by E for all inputs (E is a constant determined at compile time, depending only on the bounds X_i). For a particular input, the filter fails if $|\mathcal{E}| \leq E$, otherwise the sign of $\mathcal{E}$ is known safely.

Semi-static: Sometimes it is impossible to specify a good bound on the entries, but there is a simple formula $E = E(x_1, \ldots, x_n)$ having a structure similar to $\mathcal{E}$ that gives a valid error bound for a particular input, even when E is evaluated with single precision. E is computed dynamically (at run time) from the bounds X_i (also computed dynamically), and the filter fails if $|\mathcal{E}| \leq E$; otherwise the sign of $\mathcal{E}$ is known safely.

Dynamic: the computation of E is carried along with the computation of $\mathcal{E}$. Typically, for each arithmetic operation of $\mathcal{E}$, a rule determines the error bound for the result of that operation based on the operands and error bounds on them.

Static filters are implemented for instance in LN [11], semi-static in [1, 5], and dynamic filters in Expr [19] and LEDA [6]. Also Schewchuk, in [18], approximates $\mathcal{E}$ up to first order error terms, then up to second order errors etc., until the sign can be safely determined, this procedure combines a dynamic filter according to our description with exact computation, since it can reduce the error to zero if needed. A static filter does the floating point evaluation of

[1] Single precision here means a precision of 53 bits, used by IEEE 754 doubles.

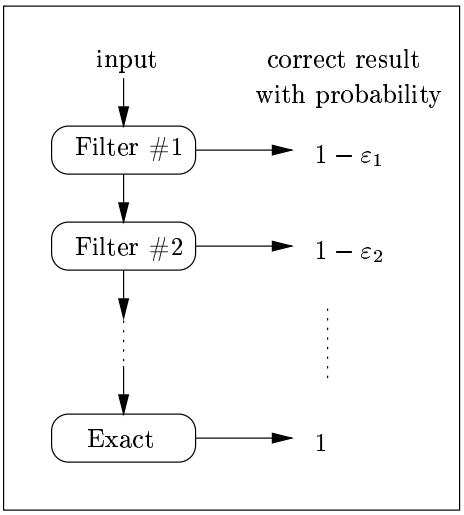

Figure 2: *Schema of cascaded filters: increasing cost and probability of success. ε_i is the probability of failure of filter # i.*

$\mathcal{E}$ plus one extra comparison with $\mathcal{E}$, whose running time is usually negligible. The cost of a semi-static filter exceeds that of a static filter by the cost of computing the error bound E, which is typically about as much as for the computation of $\mathcal{E}$. Finally, the cost of a dynamic filter is even worse, but as this is the more powerful filter (best probability of success because of more precise error computations, and usability of all operations $+, -, \cdot, /, \sqrt{}$), we tried to find a good implementation for this kind of filter.

More complicated and integrated schemes involve several filters of increasing cost and probability of success (Fig. 2) [9, 5]. Exact computation can be considered the last filter, which never fails. The cost of the total computation can then be expressed as a combination of the costs of the different filters multiplied by their conditional probability of success.

Remark : Many geometric predicates boil down to computing the sign of a determinant. Much effort has already been made towards the exact evaluation of signs of determinants, using various specific solutions such as Clarkson's or the lattice method [8, 1, 4], or using general solutions such as exact integer arithmetic [10] and modular arithmetic [2]. For $d \times d$ determinants, the complexities range from $O(d^3 \log d)$ to $O(d^4 \log d)$ with a potentially large constant in the asymptotic bounds. Practically, all these methods are several orders of magnitude slower than the straightforward, inexact floating point evaluation. In [3], we detail the higher level algorithms for computing signs of high dimensional determinants, using the interval techniques described in this paper.

We now describe in detail a particular filter, which is dynamic, based on interval arithmetic.

3 Interval arithmetic

The major tool used within our filter is *interval arithmetic*. The use of interval arithmetic in the context of matrix operations was originally proposed by Moore [16] and further promoted through a research group directed by Kulisch (see [12] for a recent survey of the available computational methods). In [12], interval arithmetic is successfully applied to many basic computation tasks in numerical linear and nonlinear algebra.

3.1 Definition

Interval arithmetic deals with *intervals* $[x] = [\underline{x}, \overline{x}]$ of (possibly infinite) reals, whose bounds are stored in double precision floating point numbers. These intervals may arise from uncertainty in the input (e.g., when the input is subject to imprecise measurements) as well as from approximate intermediate calculations. In fact, our interval methods can be applied when the input variables are intervals but in our applications they are given exactly. This is because we want to use interval arithmetic as a preliminary stage for *exact* computation. Note that for interval–type input matrices e.g. the determinant does not necessarily have a unique sign; this fact cannot be altered if we later use exact arithmetic.

The basic interval operations are defined essentially as in [12]. Namely, if both operands $[x] = [\underline{x}, \overline{x}]$, $[y] = [\underline{y}, \overline{y}]$ are (possibly infinite) intervals, all arithmetic operations ($\mathcal{OP}$) must preserve the following inclusion property (X and Y are (possibly infinite) reals) :

$$\forall\, X \in [x], \forall\, Y \in [y], (X\ \mathcal{OP}\ Y) \in ([x]\ \mathcal{OP}\ [y])$$

so we extend the operators as follows :

$$
\begin{array}{lcl}
[x] + [y] & = & [\underline{x} \mathbin{\underline{+}} \underline{y}, \overline{x} \mathbin{\overline{+}} \overline{y}] \\
[x] - [y] & = & [\underline{x} \mathbin{\underline{-}} \overline{y}, \overline{x} \mathbin{\overline{-}} \underline{y}] \\
[x] \cdot [y] & = & [\min\{\underline{x} \mathbin{\underline{\cdot}} \underline{y}, \underline{x} \mathbin{\underline{\cdot}} \overline{y}, \overline{x} \mathbin{\underline{\cdot}} \underline{y}, \overline{x} \mathbin{\underline{\cdot}} \overline{y}\}, \max\{\underline{x} \mathbin{\overline{\cdot}} \underline{y}, \underline{x} \mathbin{\overline{\cdot}} \overline{y}, \overline{x} \mathbin{\overline{\cdot}} \underline{y}, \overline{x} \mathbin{\overline{\cdot}} \overline{y}\}] \\
[x]/[y] & = & \left\{ \begin{array}{lll} [\min\{\underline{x} \mathbin{\underline{/}} \underline{y}, \underline{x} \mathbin{\underline{/}} \overline{y}, \overline{x} \mathbin{\underline{/}} \underline{y}, \overline{x} \mathbin{\underline{/}} \overline{y}\}, \max\{\underline{x} \mathbin{\overline{/}} \underline{y}, \underline{x} \mathbin{\overline{/}} \overline{y}, \overline{x} \mathbin{\overline{/}} \underline{y}, \overline{x} \mathbin{\overline{/}} \overline{y}\}] & , & 0 \notin [y] \\ [-\infty, \infty] & , & \text{otherwise} \end{array} \right. \\
\sqrt{[x]} & = & \left\{ \begin{array}{lll} [\underline{\sqrt{}}\max(\underline{x}, 0)), \overline{\sqrt{}}\ \overline{x}] & , & 0 \le \overline{x} \\ undefined & , & \text{otherwise} \end{array} \right.
\end{array}
$$

The underlined and overlined operators ($\underline{+}, \overline{+}, \underline{/} \ldots$) correspond to the IEEE 754 operations on doubles with rounding mode set to $-\infty$ or ∞.

3.2 Implementation of basic interval operations

We implemented interval operations using IEEE standard 754 rounding, with two interfaces, the first one using C++ operators, the other one using C macros. Since we use the symmetry property of IEEE 754 ($a \underline{+} b = -((-a) \overline{+} (-b))$ and similar rules), we maintain here the upper interval bound and the *negative* of the lower interval bound, so that we can always round upwards, except for square roots. For example, if $[x] = [\underline{x}, \overline{x}]$ and $[y] = [\underline{y}, \overline{y}]$ we compute the smallest interval $[z] = [\underline{z}, \overline{z}]$ such that $[x] + [y] \subset [z]$ by setting $(-\underline{z}) = (-\underline{x}) \overline{+} (-\underline{y})$ and $\overline{z} = \overline{x} \overline{+} \overline{y}$. Substraction is implemented very similarly. Multiplications and divisions are slightly more complex because they require some case distinctions for the various possible signs of $\underline{x}, \overline{x}, \underline{y}, \overline{y}$. The advantage of taking the negative of the lower interval bounds is that within a sequence of operations the rounding mode never has to be adjusted, if initially set to $+\infty$.[2]

For a geometric predicate this means that the rounding mode is set manually *outside* the interval operators, at the beginning of the computation and reset afterwards to avoid side effects. We call this implementation "OutFilter". We also provided an implementation called "InFilter" that is safer against misuse since here the rounding mode is automatically set and reset *within* each operator call, but it is of course slower.

The following piece of C code illustrates our technique. Here we consider that the rounding mode is already set to $+\infty$.

[2]Note that on Alpha processors, it might be more interesting to keep rounding towards $-\infty$ instead, which allows to never change a general rounding register. The mechanism in this case is totally equivalent.

```
typedef struct {
  double i, s; /* i stores the negative of the lower bound */
} _DIS;

inline _DIS _DIS_add (_DIS a, _DIS b)
{
  _DIS res;
  res.s = a.s + b.s;
  res.i = a.i + b.i;
  return res;
}
```

Without the trick using the opposite of the lower bound, we would have had something like BIAS:

```
VOID BiasAddII (const PBIASINTERVAL pR,
                const PBIASINTERVAL pA,
                const PBIASINTERVAL pB)
{
  _BiasRoundDown ();
  pR->inf = pA->inf + pB->inf;
  _BiasRoundUp ();
  pR->sup = pA->sup + pB->sup;
  _SetRoundToNearest();
}
```

Changing the rounding mode is basically one machine instruction, if done in assembly, but it is costly compared to e.g. one addition, since it breaks the pipeline, which is a major bottleneck on modern processors. Moreover, on some processors (e.g. UltraSparc), the interval addition can be parallelized in one cycle, which is not possible with the traditionnal scheme. These are the reasons why we wanted to avoid it. Moreover, this scheme allows to use C code (easier to port and implement than assembly) to change the rounding mode with a lesser performance penalty, it might even be tolerable in some cases.

NaNs, Overflow, Underflow :

NaNs can occur, but they are treated in the comparison functions, just like a failure of the filter. The overflow does not need particular treatment, because $-\infty$ and $+\infty$ are part of the IEEE standard 754. Underflow is not a problem here, as opposed to other dynamic filters, such as the one used in [5], consider the following example :

```
Dis a = .1;  // Interval C++ type
for (int i=0; i<10; i++) {
  a = a * a;
  cout << a << endl;
}
```

We get the following output:

```
[0.01;0.01]
[1e-04;0.0001]
[1e-08;1e-08]
```

```
[1e-16;1e-16]
[1e-32;1e-32]
[1e-64;1e-64]
[1e-128;1e-128]
[1e-256;1e-256]
[0;4.94066e-324]
[0;4.94066e-324]
```

We can see that it doesn't degenerate to the interval $[0;0]$.

Comparisons : In our C++ implementation in the CGAL library, the comparison operators ($\mathcal{COP}$ is $<, != ...$) have the following property :

$$[x]\ \mathcal{COP}\ [y] \iff (\forall\ X \in [x], \forall\ Y \in [y], X\ \mathcal{COP}\ Y)$$

So they only return safe answers. Otherwise, it means we cannot certify the comparison, the roundoff error was too large, and we throw an exception. This exception is caught inside the predicate function, and the computation is done again but with an exact type.

3.3 Discussion

It turns out that the interval evaluation of expressions is not always effective, depending on the particular structure of the expression. This notion of effectiveness is related to the relative size of the resulting intervals, not to the time necessary to evaluate the interval expression. This is because the interval evaluation incurs only a constant overhead over the usual floating–point approximation[3]. Important types of expressions that are well suited for interval evaluation are *dot products* or *inner products* of vectors and the derived operations of matrix-matrix product and matrix-vector product. The following *interval degree Ideg*$(\mathcal{E}) \in \mathbb{Z}$ is a heuristic, asymptotic measure for the average number of uncertain bits of $[\mathcal{E}]$ and hence for the quality of the interval evaluation of $\mathcal{E}$. For an expression $\mathcal{E}$ consisting of a single input number z we set $\mathrm{Ideg}(\mathcal{E}) = 0$ and inductively, if $\mathcal{E}$ is computed from expressions $\mathcal{X}$, $\mathcal{Y}$ by the operations $\{+, -, *, /, \sqrt{\ }\}$ we set

$$\begin{aligned}
\mathrm{Ideg}(\mathcal{X}+\mathcal{Y}) &= \max\{\mathrm{Ideg}(\mathcal{X}), \mathrm{Ideg}(\mathcal{Y})\} \\
\mathrm{Ideg}(\mathcal{X}-\mathcal{Y}) &= \max\{\mathrm{Ideg}(\mathcal{X}), \mathrm{Ideg}(\mathcal{Y})\} \\
\mathrm{Ideg}(\mathcal{X}\cdot\mathcal{Y}) &= 1+\max\{\mathrm{Ideg}(\mathcal{X}), \mathrm{Ideg}(\mathcal{Y})\} \\
\mathrm{Ideg}(\mathcal{X}/\mathcal{Y}) &= 1+\max\{\mathrm{Ideg}(\mathcal{X}), \mathrm{Ideg}(\mathcal{Y})\} \\
\mathrm{Ideg}(\mathcal{X}^{1/2}) &= \mathrm{Ideg}(\mathcal{X}).
\end{aligned}$$

In this notation, the mentioned products all have degree 1. On the other hand, many of the basic operations in linear algebra have larger degree. For example, the degree of computing $\det(A)$, the degree of computing decompositions $P \cdot A = L \cdot U$, and the degree of solving triangular systems for a d dimensional matrix are all $\Theta(d)$. Typically, the computed intervals are useless if $\mathrm{Ideg}(\mathcal{E})$ has the same order of magnitude than the used mantissa length p of the floating–point numbers and are very useful if $\mathrm{Ideg}(\mathcal{E})$ is a small constant. As an example for the calculation of the degree we prove the following lemma.

Lemma 1 *The interval solution of a triangular system $T \cdot [x] = b$ with exact input data has degree d.*

[3]This overhead depends on the particular platform given by hardware architecture, programming language and compiler. We obtain a factor from 2 to 6 with our implementation.

Proof: Without loss of generality, we consider forward substitution with a lower triangular matrix $T = (t_{i,j})_{i,j}$. In the basic step $d = 0$ we have $x_0 = b_0/t_{0,0}$ which has degree 1. Let for $d > 0$ $x_0, \ldots, x_{d-1}$ have degree $\leq d$ and $\mathrm{Ideg}(x_{d-1}) = d$. From the formula

$$x_d = b_d/l_{d,d} - \sum_{j=0}^{d-1} x_j(l_{d,j}/l_{d,d})$$

we see that since $\mathrm{Ideg}(l_{d,j}/l_{d,d}) = 1$, each product $x_j(l_{d,j}/l_{d,d})$ has degree $\leq d+1$. Because the latter term has degree $d+1$ for $j = d$, the whole sum has degree $d+1$. Our statement follows by induction. ■

Experiments with a randomly chosen matrix A show that the interval solution of a system $A[x] = b$ using a LU decomposition of A in fact incurs an uncertainty in the last $\Theta(d)$ places, even if A is tridiagonal. Another simple lemma is that Gauss elimination is of degree $2d$.

Let us stress again that our notion of degree gives a *purely heuristic and asymptotic* estimate of the usefulness of interval arithmetic. Our point is that interval arithmetic can be useful for expressions with small bounded degree or else if the problem has small dimension.[4] Nevertheless, our degree is logarithmically related to the index of [5] (if we do not count additions), and this index gives a bound on the relative error of a computation.

4 Experimental Validation

We present our filter in a variety of settings encountered by geometric algorithms. In this paper, we did not investigate the practical efficiency of our filters for solving linear problems.

Isolated primitive operations. This section presents some benchmarks on isolated basic operations in loops. Timings are in seconds, for 10^8 loops on a 300MHz UltraSparc, compiled with the EGCS compiler. We compare the following implementations :

- BIAS: the Profil/BIAS library [20], written in C.
- C++ (in): our method, with rounding mode changes packed into the operators.
- C++ (out): our method, with rounding mode changes outside the operators, as it can be used for geometric predicates.
- Macro: same thing, but using a macro interface, that seems to help the compiler to optimize.
- Assembly: the previous assembly code generated, easily optimized by hand, so that we can see the potential of the method, when the compiler is able to fully optimize. Only done for the addition.
- FP: the straightforward floating point code.

We observe that a lot of time is spent doing useless copies of the objects (C++ copy constructor). It seems that improving the compiler would yield much better results for our number type. However, even with this large room for improvement, the results look good, especially concerning the addition.

[4] For problems of large degree in large dimensions, interval arithmetic can still be useful if the input data has an appropriate (sparse) structure.

	+	*	/
BIAS	10.2	14.2	31.4
C++ (in)	16.6	33.3	44.5
C++ (out)	4.4	14.5	28.6
Macro	3.4	4.8	18.9
Assembly	1.0	-	-
FP	1.0	1.0	7.5

Table 1: Running time of primitive operations in loops, using different implementation approaches.

Isolated geometric predicates. This section presents some benchmarks on isolated well known low-dimensional predicates, to show the overhead caused by the use of various filters, compared to the pure floating point computation.

We evaluate the semi-static filters given in [5], the semi-static error bound being evaluated once only for the entire predicate in "Semi-static," and once for each operation in "FpFilter."

Beside our new filter types "OutFilter" and "InFilter" we also tested two other, completely different dynamic filters called "AbsFilter" and "RelFilter" that use standard error analysis to dynamically compute absolute and relative errors, respectively.

The considered predicates are the standard orientation and in-sphere predicates, and a specific in-circle predicate for the Voronoi diagram of line segments that asks whether a point is in the circle tangent to two lines and passing through a point site. The latter predicate involves three square root operations but no divisions. The benchmarks of table 2 were made on a Ultra Sparc 200 MHz, with the GNU compiler using the flag -O2.

	Insphere 3D	Orient 3D	Orient 2D	Incircle 2D (llp_p)
Semi-static	2.28	1.63	1.45	1.58
FpFilter	3.36	2.55	1.77	2.17
OutFilter	5.15	2.98	1.92	3.38
InFilter	10.58	5.85	3.13	4.89
AbsFilter	20.00	10.68	5.38	7.42
RelFilter	94.00	102.5	69.70	9.23

Table 2: Running time overhead caused by the use of various filters, compared to the static filter (pure floating-point computation). The time taken by other computations in case of filter failure is not taken into account.

As our experiments show, the variant "OutFilter" is always the fastest. Hence in all other experiments, we use this filter variant only. Table 2 documents only differences in the running times, without accounting for the other computations in case of failure. Moreover, the filters have different probabilities of success. For a fair comparison, one needs to evaluate them within some geometric algorithm.

Geometric algorithms. Tested on a 2 dimensional Delaunay triangulation of random points, the cost of using filtered predicates with our generic scheme, compared to the traditional use of doubles, appears to be about 50%. This particular algorithm spends about 30% of its time in the predicates (*orientation* mainly, and marginally *in_sphere*), when using doubles. Depending on the algorithm, the overhead can be a little bit more or less, but anyway it can be acceptable for a generic solution.

5 Conclusions

The best prior art proposes static filters to low-dimensional primitives dealing only with integer entries [1, 4, 8, 11], or to rational $2d$ and $3d$ primitives when dealing with floating-point entries [18]. Our solution is powerful as it avoids the overestimation of errors induced by static bounds and is efficient, i.e. always less than an order of magnitude slower than the straightforward floating-point implementation. Unlike previous filters, it can handle square roots and divisions. Square roots are needed for instance in computing the Voronoi diagram of a set of points and line segments, and divisions might be needed in the efficient computation of the sign of a large determinant. For the latter problem, our filter is the first that works for arbitrary dimensions.

For low-dimensional geometry, we have packaged this filter for the CGAL C++ library [7]. There is also a C implementation available at [21]. Experiments show that it is only little slower than the semi-static filter when used in a predicate and with a precompiler such as [5], but that it rarely fails on non-degenerate instances that make the semi-static filter fail. Hence, we recommend interval arithmetic as the ultimate level of filter before resorting to exact arithmetic. In most cases, we expect that resorting to exact arithmetic will not be needed.

Plans for the future include the implementation of filtered precomputations of temporary geometric objects. For example, many instances of the in_sphere tests with the same circle but different test points may benefit from precomputing this circle (in interval representation). These optimizations are currently not possible within the paradigm that makes robustness relying only on exact predicates.

References

[1] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluating signs of determinants using single-precision arithmetic. *Algorithmica*, 17:111–132, 1997.

[2] H. Brönnimann, I. Emiris, V. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, 1997.

[3] H. Brönnimann, C. Burnikel and S. Pion. Interval Arithmetic Yields Efficient Dynamic Filters for Computational Geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1998.

[4] H. Brönnimann and M. Yvinec. Efficient exact evaluation of signs of determinants. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 166–173, 1997.

[5] C. Burnikel, S. Funke, and M. Seel. Exact arithmetic using cascaded computations. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 175–183, 1998.

[6] C. Burnikel, J. Könnemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995.

[7] The CGAL Consortium. Computational Geometry Algorithms Library http://www.cs.uu.nl/CGAL/

[8] K. L. Clarkson. Safe and effective determinant evaluation. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 387–395, 1992.

[9] O. Devillers and F. P. Preparata. A probabilistic analysis of the power of arithmetic filters. Technical Report CS-96-27, Center for Geometric Computing, Dept. Computer Science, Brown Univ., 1996.

[10] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, 1993.

[11] S. Fortune and C. J. van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometr y. *ACM Trans. Graph.*, 15(3):223–248, July 1996.

[12] R. Hammer, M. Hocks, U. Kulisch, and D. Ratz. *C++ Toolbox for Verified Computing.* Springer, 1995.

[13] IEEE. IEEE standard 754-1985 for binary floating-point arithmetic. Reprinted in SIGPLAN *Notices* 22(2):9–25, 1987.

[14] R. B. Kearfott. Interval Computations: Introduction, Uses, and Resources. 1996.

[15] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *13th World Computer Congress IFIP94*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.

[16] R.E. Moore. *Interval Analysis.* Prentice-Hall, 1966.

[17] A. Neumaier. *Interval Methods for Systems of Equations.* Cambridge University Press, 1990.

[18] J. R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[19] C. K. Yap and T. Dubé. The exact computation paradigm. In *Computing in Euclidean Geometry*, D.-Z. Du and F. K. Hwang, eds., Lecture Notes Series on Computing, volume 1, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.

[20] O. Knüppel. The PROFIL/BIAS library. http://www.ti3.tu-harburg.de/Software/PROFILEnglisch.html

[21] S. Pion. Interval Arithmetic Library. http://www.inria.fr/prisme/personnel/pion/progs/interval